# OpenMP Prefix Sum and MPI Finding an element in a large array

Harsh Kumar, 24576

21 November 2024

---

1. Q1

**Ans:** We have solved the prefix problem for integer arrays of sizes 10,000, 20,000, and 30,000 elements using different thread counts (2, 4, 8, 16, 32, and 64). To ensure accuracy and consistency, we ran the code five times for each configuration and recorded the average execution time. We also calculated speedups relative to sequential execution and visualized the results through graphs.

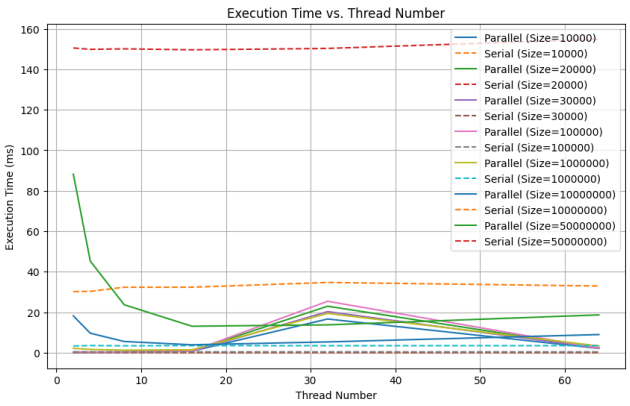
**Methodology for Parallel Program**

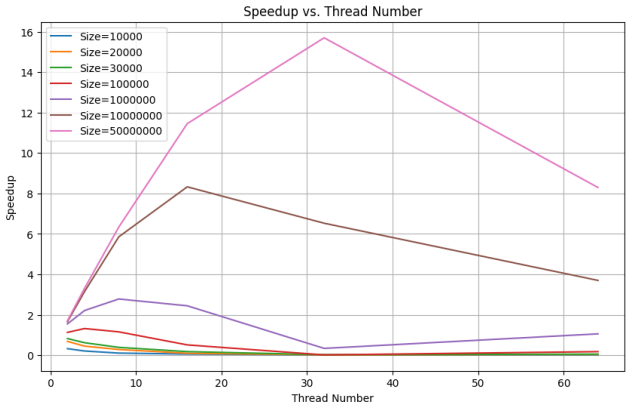
- Array Division: The input array A[] is divided into blocks. Each block is assigned to a different OpenMP thread based on its thread ID.
- Local Computation: Each thread computes the prefix sum of its assigned block. Specifically, the B[] array for each thread is calculated, with the result of each block's last element stored as a partial sum.
- Global Adjustment: After all threads compute their local prefix sums, an additional step is performed by a single thread to adjust the prefix sums for threads based on the partial sums. This step ensures that the results from different threads are correctly combined.
- Final Calculation: The prefix sum from the partial sums array is then propagated across the entire array B[] using mutiple threads.

**Experiments :**
The program was executed for various combinations of:

- **Array sizes:** 10000, 20000, 30000, 100000, 1000000, 10000000, 50000000. (Tested on extra sizes for showcasing speed up, as the size given in question were not providing good speedup)
- **Thread counts:** 2, 4, 8, 16, 32, 64.
- **Repetitions:** Each configuration was run 5 times, and the average execution time was recorded.

**Results**






**Data**

| Size | Thread Num | Parallel Time (ms) | Serial Time (ms) | Speedup |
|---|---|---|---|---|
| 10000 | 2 | 0.127304 | 0.040490 | 0.325840 |
| 10000 | 4 | 0.201487 | 0.042146 | 0.210099 |
| 10000 | 8 | 0.405309 | 0.041840 | 0.106444 |
| 10000 | 16 | 0.757608 | 0.045308 | 0.059818 |
| 10000 | 32 | 16.631042 | 0.042970 | 0.016695 |
| 10000 | 64 | 2.130471 | 0.044910 | 0.021094 |
| 20000 | 2 | 0.119120 | 0.079314 | 0.684536 |
| 20000 | 4 | 0.190678 | 0.085230 | 0.451163 |
| 20000 | 8 | 0.283442 | 0.078167 | 0.282820 |
| 20000 | 16 | 0.863872 | 0.082994 | 0.099077 |
| 20000 | 32 | 22.950946 | 0.083362 | 0.018574 |
| 20000 | 64 | 2.269389 | 0.083534 | 0.037170 |
| 30000 | 2 | 0.144250 | 0.117286 | 0.825022 |
| 30000 | 4 | 0.195026 | 0.119916 | 0.619767 |
| 30000 | 8 | 0.329071 | 0.123604 | 0.389613 |
| 30000 | 16 | 0.701772 | 0.123556 | 0.180435 |
| 30000 | 32 | 20.271132 | 0.124114 | 0.025076 |
| 30000 | 64 | 2.275345 | 0.125834 | 0.055445 |

Table 1: Performance Metrics for Different Problem Sizes and Thread Numbers (given in question)

| Size | Thread Num | Parallel Time (ms) | Serial Time (ms) | Speedup |
|---|---|---|---|---|
| 50000000 | 8 | 23.674024 | 150.089310 | 6.340838 |
| 50000000 | 16 | 13.061660 | 149.592113 | 11.453240 |
| 50000000 | 32 | 13.745516 | 150.292523 | 15.697788 |
| 50000000 | 64 | 18.663620 | 154.784480 | 8.296378 |
| 10000000 | 4 | 9.614381 | 30.302174 | 3.151940 |
| 10000000 | 8 | 5.531018 | 32.336211 | 5.855866 |
| 10000000 | 16 | 3.888767 | 32.343829 | 8.328022 |
| 10000000 | 32 | 5.315737 | 34.679534 | 6.528956 |

Table 2: Best Speedups for Different Problem Sizes and Thread Numbers

**For the complete table and data please refer to the third section.**

**Observations:**

- Small Problem Sizes: Parallelism is inefficient for small arrays, as threading overhead outweighs computation benefits, leading to poor or negative speedups.

- Large Problem Sizes: Speedup improves significantly for larger arrays with higher thread counts, but again diminishes due to overhead and contention.
- Best Performance: Maximum speedup (15.7x) is achieved for the largest problem size (50,000,000 elements)
- Excessive thread counts (32–64) generally degrade performance for our problem due to synchronization overhead.

2. Q2

**Ans:** We have implemented an MPI-based parallel search algorithm that can find a target element in a distributed array across multiple processes. The steps are as follows:

1. **Initialization:**
   - We initialize MPI and obtain the rank and size of the processes.
   - Root process (`rank 0`) generates a random integer array of size 1,000,000 with values between 1 and 5,000,000.
2. **Array Distribution:**
   - The array is distributed across processes using `MPI_Scatter`.
3. **Broadcast Target:**
   - A target integer is broadcast to all processes using `MPI_Bcast`.
4. **Local Search:**
   - Each process searches its local array for the target element.
   - If found, the global index is calculated and other processes are notified via `MPI_Send`.
5. **Termination Notification:**
   - Processes use `MPI_Irecv` to listen for termination signals and stop searching upon receipt.
   - Processes does not locate the target immediately, it checks for messages from other processes using the MPI_Test function after each 1000 elements ( to reduce communication overhead).
     - If received, Update the global find and exit the loop
6. **Result Aggregation:**
   - The global index of the target is collected using `MPI_Allreduce` and displayed by the root process.
7. **Performance Measurement:**
   - Execution times are recorded for configurations of 1, 8, 16, 32, and 64 processes.
   - 20 random targets are tested for each configuration to calculate average execution time.
8. **Cleanup:**
   - Memory is cleaned up and MPI is finalized.

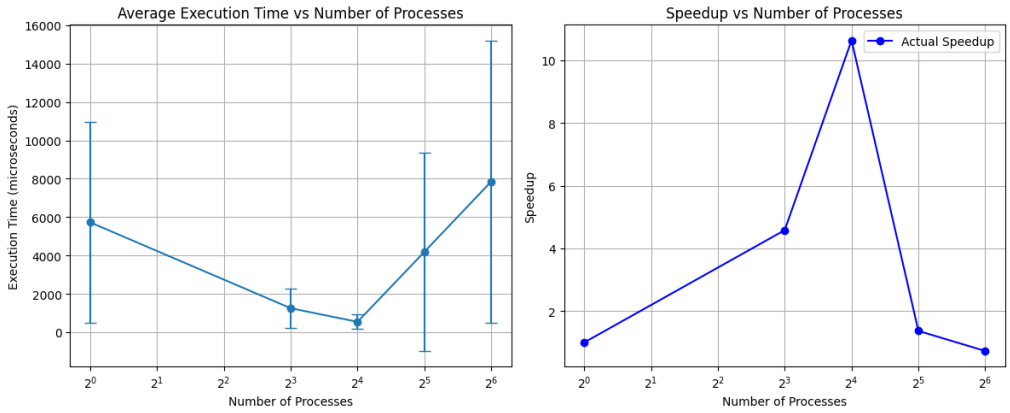
## Results

Execution times and speedup for various numbers of processes are summarized below:

| Processes | Avg Time ($\mu$s) | Std Dev | Speedup |
|---|---|---|---|
| 1 | 5726.90 | 5251.40 | 1.00 |
| 8 | 1250.00 | 1015.23 | 4.58 |
| 16 | 538.55 | 382.43 | 10.63 |
| 32 | 4184.90 | 5165.66 | 1.37 |
| 64 | 7853.45 | 7360.73 | 0.73 |

Table 3: Performance Summary

### Speedup graphs




## Observations

- The program achieves significant speedup up to 16 processes due to effective load distribution.
- Beyond 16 processes, communication overhead reduces the benefits of parallelism.

## Conclusion

The MPI-based search program demonstrates scalability for up to 16 processes, showcasing efficient parallelism within the limits of communication overhead.

3. Complete Data for Question 1

**Ans:** PTO

| | size | thread_num | parallel_time | serial_time | speedup |
|---|---|---|---|---|---|
| 0 | 10000 | 2 | 0.127304 | 0.040490 | 0.325840 |
| 1 | 10000 | 4 | 0.201487 | 0.042146 | 0.210099 |
| 2 | 10000 | 8 | 0.405309 | 0.041840 | 0.106444 |
| 3 | 10000 | 16 | 0.757608 | 0.045308 | 0.059818 |
| 4 | 10000 | 32 | 16.631042 | 0.042970 | 0.016695 |
| 5 | 10000 | 64 | 2.130471 | 0.044910 | 0.021094 |
| 6 | 20000 | 2 | 0.119120 | 0.079314 | 0.684536 |
| 7 | 20000 | 4 | 0.190678 | 0.085230 | 0.451163 |
| 8 | 20000 | 8 | 0.283442 | 0.078167 | 0.282820 |
| 9 | 20000 | 16 | 0.863872 | 0.082994 | 0.099077 |
| 10 | 20000 | 32 | 22.950946 | 0.083362 | 0.018574 |
| 11 | 20000 | 64 | 2.269389 | 0.083534 | 0.037170 |
| 12 | 30000 | 2 | 0.144250 | 0.117286 | 0.825022 |
| 13 | 30000 | 4 | 0.195026 | 0.119916 | 0.619767 |
| 14 | 30000 | 8 | 0.329071 | 0.123604 | 0.389613 |
| 15 | 30000 | 16 | 0.701772 | 0.123556 | 0.180435 |

| | | | | | |
|---|---|---|---|---|---|
| 16 | 30000 | 32 | 20.271132 | 0.124114 | 0.025076 |
| 17 | 30000 | 64 | 2.275345 | 0.125834 | 0.055445 |
| 18 | 100000 | 2 | 0.375477 | 0.405741 | 1.130139 |
| 19 | 100000 | 4 | 0.324188 | 0.419853 | 1.321134 |
| 20 | 100000 | 8 | 0.366781 | 0.423689 | 1.156880 |
| 21 | 100000 | 16 | 0.812724 | 0.410131 | 0.512629 |
| 22 | 100000 | 32 | 25.389135 | 0.412759 | 0.016455 |
| 23 | 100000 | 64 | 2.228281 | 0.398051 | 0.178756 |
| 24 | 1000000 | 2 | 2.137081 | 3.310159 | 1.550438 |
| 25 | 1000000 | 4 | 1.600611 | 3.509708 | 2.212294 |
| 26 | 1000000 | 8 | 1.242307 | 3.408323 | 2.782206 |
| 27 | 1000000 | 16 | 1.427149 | 3.475516 | 2.445694 |
| 28 | 1000000 | 32 | 19.400570 | 3.441069 | 0.339733 |
| 29 | 1000000 | 64 | 3.303789 | 3.469768 | 1.058547 |
| 30 | 10000000 | 2 | 18.165632 | 30.152856 | 1.660598 |
| 31 | 10000000 | 4 | 9.614381 | 30.302174 | 3.151940 |
| 32 | 10000000 | 8 | 5.531018 | 32.336211 | 5.855866 |
| 33 | 10000000 | 16 | 3.888767 | 32.343829 | 8.328022 |
| 34 | 10000000 | 32 | 5.315737 | 34.679534 | 6.528956 |
| 35 | 10000000 | 64 | 8.950479 | 32.937430 | 3.701584 |
| 36 | 50000000 | 2 | 88.155199 | 150.507101 | 1.707388 |
| 37 | 50000000 | 4 | 45.229633 | 149.824115 | 3.312820 |
| 38 | 50000000 | 8 | 23.674024 | 150.089310 | 6.340838 |
| 39 | 50000000 | 16 | 13.061660 | 149.592113 | 11.453240 |
| 40 | 50000000 | 32 | 13.745516 | 150.292523 | 15.697788 |
| 41 | 50000000 | 64 | 18.663620 | 154.784480 | 8.296378 |